IN THE SPECIFICATION

Please amend the paragraphs below to read as follows:

[0001] This invention relates to computer processing systems, and particularly controls entry insertion into a Branch Target Buffer (BTB) table via a queue structure which also serves the purpose of creating synchronization between asynchronous branch prediction and instruction decoding [[e]] to overcome start-up latency effects in a computer processing system.

[0002] A basic pipeline microarchitecture of a microprocessor processes one instruction at a time. The basic dataflow for an instruction follows the steps of: instruction fetch, decode, address computation, data read, execute, and write back. Each stage within a pipeline or pipe occurs in order; and hence a given stage can not progress unless the stage in front of it is progressing. In order to achieve highest performance for the given base, one instruction will enter the pipeline every cycle. Whenever the pipeline has to be delayed or cleared, this adds latency which in turn can be monitored by evaluating the performance of a microprocessor carrying [[ies]] out a task. While there are many complexities that can be added on to such a pipe design, this sets the groundwork for branch prediction theory related to the present ~~stated~~ invention.

[0003] There are many dependencies between instructions which prevent the optimal case of a new instruction from entering the pipe every cycle. These dependencies add latency to the pipe. One category of latency contribution deals with branches. When a branch is decoded, [[is]] it can either be "taken" or "not taken." A branch is an instruction which can either fall through to the next sequential instruction, that is "not taken," or branch off to another instruction address, that is, "taken," and carries out execution on a different sequence of code.

[0004] At decode time, the branch is detected, and must wait to be resolved in order to know the proper direction in which the instruction stream is to proceed. By waiting for potentially multiple pipeline stages for the branch to resolve the direction in which to proceed, latency is added into the pipeline. To overcome the latency of waiting for the branch to resolve, the direction of the branch can be predicted such that the pipe begins decoding either down the "taken" or "not taken" path. At branch resolution time, the guessed direction is compared to the actual direction the branch was to take. If the actual direction and the guessed direction are the same, then the latency of waiting for the branch to resolve has been removed from the pipeline in this scenario. If the actual and predicted direction miscompare, then decoding has proceeded down the improper path and all instructions in this improper path behind that of the improperly guessed direction of the branch must be flushed out of the pipe. [[, and]] Then the pipe must be restarted at the correct instruction address to begin decoding the actual path [[of]] the given branch is supposed to take.

[0005] Because of controls involved with flushing the pipe and beginning over, there is a penalty associated with the improper guess and latency is added into the pipe over simply waiting for the branch to resolve the issue of the correct path before decoding further. By having a proportionally higher rate of correctly guessed paths, the ability to remove latency from the pipe by guessing the correct direction [[out]] outweighs the latency added to the pipe for guessing the direction incorrectly.

[0006] In order to improve the accuracy of the ~~guesses associated with the~~ a guess of a branch, a Branch History Table (BHT) can be implemented which allows for direction guessing of a branch based on the past behavior of the direction the branch previously went. If the branch is always taken, as is the case of a subroutine return, then the branch will always be guessed as taken. IF/THEN/ELSE structures ~~become~~ are more complex in their behavior. A branch may be always taken, sometimes taken and not taken, or always not taken. Based on the implementation of a dynamic branch predictor, this will determine how well the BHT table predicts the direction of the branch.

[0007] When a branch is guessed taken, the target of the branch is to be decoded. The target instruction of the branch is acquired by making a fetch request to the instruction cache for the address which is the target of the given branch. Making the fetch request out to the cache involves minimal latency if the target address is found in the first level of cache. If there is not a hit in the first level of cache, then the fetch continues through the memory and storage hierarchy of the machine until the instruction text for the target of the branch is acquired. Therefore, any given taken branch detected at decoding [[e]] has a minimal latency associated with it that is added to the amount of time it takes the pipeline to process the given instruction. Upon missing a fetch request in the first level of memory hierarchy, the latency penalty the pipeline pays grows higher and higher, the ~~further~~ farther up in the hierarchy the fetch request must progress, until a hit occurs. In order to hide part or all of the latency associated with the fetching of a branch target, a BTB table ~~branch target buffer(BTB)~~ can work in parallel with a BHT table.

[0008] Given a current address which is currently being decoded from an input, the BTB table can search for the next instruction address from this point forward which contains a branch. Along with storing the instruction address of branches in the BTB table, the target of the branch is also stored with each entry. With the target being stored, the address of the target can be fetched before the branch is ever decoded. By fetching the target address ahead of decoding, [e,]] latencies associated with cache misses can be minimized to the point of time it takes between the fetch request and the ~~decode~~ decoding of the ~~branch's~~ target of the branch.

[0009] In designing a BTB table, the ~~amount~~ number of branches that can be stored therein [[it]] is part of the equation that determines how beneficial the structure is. In general, a BTB table is indexed by part of an instruction address within the processor, and tag bits are stored in the BTB table such that the tag bits must match the remaining address bits of concern that were not used for the indexing. In order to improve the efficiency of the BTB table, it can be created such that it has an associativity greater than one. By creating an associativity greater than one, multiple branch/target pairs can be stored for a given index into the array. To determine which is the correct entry, if an entry at all, the tag bits are used to select one entry, at most, ~~entries~~ from the multiple entries stored for a given index.

[0010] When a branch is determined at ~~decode~~ decoding time and it was not found ahead of time by the asynchronous BTB/BHT table function, the branch is determined as a surprise branch. A surprise branch is any branch which was not found by the dynamic branch prediction logic ahead of the time of decoding. [[e.]] A branch is not predicted by the branch prediction logic because it was not found in the BTB/BHT tables. There are two reasons that a branch is not found in the BTB/BHT tables. If a branch is not installed in the BTB/BHT tables, then a branch can not be found, as it is nowhere ~~no where~~ to be found. The second scenario is ~~when~~ a situation in which the branch resides in the BTB/BHT tables. [[; h]] However, not enough processing time has [[not]] been ~~presented such~~ available so that [[the]] searching could find the branch prior to ~~it being~~ decoding thereof. [[ed.]] In general, branch prediction search algorithms can have a high throughput. [[; h]] However, the latency required for starting a search can be [[of]] a reasonable length longer compared to the starting of instructions down the pipeline in respect to the time frame that an instruction decodes.

[0011] Whenever a branch is detected at decoding [[e]] time, where the branch is a surprise branch, upon knowing the target and direction of the branch at a later time, [[it]] an entry can be written into the BTB and the BHT tables. Upon writing the entry into the tables, the entry can ideally be found the next time a search is in the area in the machine of the stated branch.

[0012] In the case [[that]] in which a branch resides in the BTB/BHT tables but latency effects prevent the branch from being found in time, the branch is treated like a surprise branch as this branch is no different from a branch which is not in the tables. Upon determining the target and direction of the branch, it will be written into the BTB/BHT tables. A standard method of entering a branch into the tables is to place it in the given column (associativity) that was least recently used; thereby, keeping those branches which were most recently accessed in the tables. A reading of the columns prior to the write is not performed to check for duplicates because the ~~amount~~ number of reads that would have to be performed in addition to the normal operation would be enough to cause additional latency delays which would further hinder branches from being found so they could be predicted [[; h]] Hence, this would increase the quantity of surprise branches in a series of code. Increasing the number of surprise branches causes the performance to decrease. In order to work around the raised latency issues, a recent entry has been designed to keep track of the recent entries into the BTB table. Through the process of this queue, additional reads from the BTB table are not required. Furthermore, the size of such a queue over a duplicate array or another read port on the given array is magnitudes different in size. The space for a second full size array or an additional read port can be deemed to be so great that the area of the machine spent for such an operation can be better spent elsewhere for higher performance gains. By adding a small recent entry queue, the area of the machine is kept modest while the performance delta between a queue and additional read port is minimal, if not for the better.

[0013] One problem encountered with the BTB table in the case of multiple instantiations of a branch entry is that the multiple instantiations of the branch entry can be written into a BTB table ~~branch target buffer (BTB)~~ at a high frequency based on code looping patterns. However, this hinders the BTB table performance by removing valid entries for duplicate entries of another branch. Thus, a clear need exists for a way in which to prevent multiple instantiations of a branch entry within the BTB table ~~branch target buffer~~.

[0014] The shortcomings of the prior art are overcome and additional advantages are provided through the provision of a recent entry queue that tracks the most recent branch/target data that is stored into a ~~Branch Target Buffer (BTB)~~ BTB table. Through the usage of a BTB table recent entry queue, any new entry that is to be written into the BTB table is first checked against that of the recent entry queue. If the contained data that is to be written into the BTB table is valid in the recent entry queue, then the data is already contained in the BTB table. Given the fact that the data is already in the BTB table, the data write into the BTB table for the given entry is blocked. If the data was not blocked, then it would most likely replace other valid data when a BTB table has an associativity greater than one.

[0015] As noted above, in pipeline processors of the prior art, multiple instantiations of a branch entry could be written into a Branch Target Buffer (BTB) table at a high frequency based on code looping patterns. This reduces performance by removing valid instances for duplicate entries of another branch. This had the effect of causing the BTB table to behave as a single associative design given any amount of designed associativity.

[0016] According to the method, system, and program product described herein, by keeping track of closely associated duplicates to become entries, the monitoring structure is extended [[to]] not only to block BTB table writes, but [[to]] additionally to notify instruction decode when such an operation is to take place. In the case of the majority of duplicate entries, the occurrence is initiated by an instruction loop where the first branch was not predicted in time because of branch prediction start-up latency which thereby causes each additional iteration [[to]] not to be predicted in time. By being able to predict one iteration ~~interaction~~ of the loop, the BTB table is able to get ahead and therefore potentially to predict all future iterations of the branch point. By causing a delay in the pipeline by blocking the decoding operation, the branch prediction logic is able to get ahead thereby allowing the pipeline to run at the efficiency of which it is capable. [[of.]] Such operations are viewable by higher performance which can be viewed externally as an application completing a task in a shorter time span.

[0017] Beyond blocking repetitive data blocks from being written into the BTB table, the recent entry queue also serves the purposes of being able to delay decoding [[e]] and to perform an accelerated lookup. Whenever a branch is repeatedly taken and each iteration is not predicted by the BTB table, the recent entry queue provides the data necessary to detect this such that decoding [[e]] can be delayed, thereby allowing the branch prediction logic to catch up to the decoding [[e]] pipeline of the machine.

[0018] Finally, because the queue is a very small structure, it has a minimal latency compared to the time it takes to look up an entry in the array. [[; t]] Therefore, branch detection via the branch prediction logic can occur in [[less]] fewer time/cycles when using the recent write entry queue over the BTB table. Given such a pattern, particular tight loop prediction scenarios that hinder the BTB table can be overcome via the recent entry queue thereby making these loops predictable. Predicting such loops successfully removes latency from the pipeline thereby improving the overall performance of [[such]] a machine. These performance improvements are ~~noticed through~~ evidenced by the reduced amount of time that is required for a computer program to complete the task at hand.

[0019] The method of operating a computer, the program product for operating a computer, and the computer having a pipelined processor with a BTB table ~~branch target buffer (BTB)~~ in accordance with this invention achieves these ends by creating a recent entry queue in parallel with the BTB table. ~~branch target buffer (BTB).~~ The recent entry queue comprises a set of BTB table ~~Branch Target Buffer (BTB)~~ entries, which are organized as a FIFO queue, and preferably [[e]] a FIFO queue that is fully associative for reading.

[0020] In carrying out the ~~described~~ method of this invention, an entry to be written into the BTB table is compared against the valid entries within the recent entry queue, an entry matching an entry within the recent entry queue is blocked from being written into the BTB table. When an entry is written into the BTB table it is also written into the recent entry queue.

[0021] A further step involves searching the BTB table for a next predicted branch and evaluating the recent entry queue while the BTB table is being indexed. The recent entry queue maintains a depth up to the associativity of the BTB table, whereby while the BTB table is indexed, the recent entry queue positions are input into comparison logic. The recent entry queue depth is searched in respect to a matching branch in parallel [[to]] with searching the BTB table output, whereby the hit detect logic [[to]] supports the associativity of the BTB table. In searching the BTB table for the next predicted branch, the search strategy uses a subset of the recent entry queue as a subset of the BTB table, and preferably fast indexes recently encountered branches. A further aspect of [[the]] this invention includes searching the complete recent entry queue to block duplicate BTB table writes.

[0022] ~~A further~~ Still another aspect of the invention comprises searching the recent entry queue to detect looping branches. This may be done by comparing the branch to determine if it was recently written into the queue. A further operation includes determining if the branch is backwards branching, whereby a looping branch is detected. This includes first detecting a looping branch that is not predicted, and thereafter delaying a decoding step. [[e.]] The decoding step [[e]] is delayed until a fixed number of cycles, or until the BTB table predicts a branch.

[0023] A further aspect of the described method, system, and program product of this invention comprises [[is]] staging writes to the BTB table in the recent entry queue, including delaying a write and placing the write in the recent event queue, and also detecting a predicted branch while its BTB table write is temporarily staged in the recent entry queue.

[0026] FIG. 1 illustrates one example of a Branch Target Buffer (BTB) table layout.

[0027] FIG. 2 illustrates one example of data contained within a single entry of a BTB table and recent entry queue.

[0028] FIG. 3 illustrates one example of a microprocessor pipeline.

[0029] FIG. 4 illustrates one example of a BTB table recent entry queue with compare logic.

[0030] FIG. 5 illustrates one example of a timeline for write queue access and BTB table writing.

[0031] FIG. 6 illustrates one example of a recent entry queue modifying the BTB table hit detect logic.

[0032] FIG. 7 illustrates one example of a state machine descriptor of a recent entry queue delaying decoding. [[e]

[0034] The present invention is directed to a method and apparatus for implementing a recent entry queue which complements a Branch Target Buffer (BTB) BTB table 100 as shown generally in ~~Figure~~ FIG. 1. FIG. 4 shows an example of a BTB table recent entry queue with compare logic 400. In FIG. 4 a set of data lines 410 supply data to a first set of recent entry queue registers 420 for a first entry and to a second set of recent entry queue registers 421 for a second entry. The registers 420 and 421 store branch tag, target address and valid data. The data in the first set of registers 420 is compared by a first compare equal comparator 430 with the data being supplied by data in lines 410. The data in the second set of registers 421 is compared by a second compare equal comparator 431 with the data being supplied by data in lines 410. When the compare equal comparators detect equality an output is suppied from either or both of them to the OR gate 440 which supplies a block write signal on line 450. Through the usage of a BTB table recent entry queue with compare logic 400, as shown in ~~Figure~~ FIG. 4, three benefits are acquired as follows:

1) Removal of majority of scenarios that can cause duplicate entries in [[the]] a BTB table 100 of FIG. 1, BTB table 610 in FIG. 6.

2) The ability to semi-synchronize the asynchronous interface between branch prediction and decoding [[e]] when the latency of the branch detection via the BTB table initially places the BTB table behind the decoding [[e]] of a pipeline when the pipeline is starting up from a cold start or after a branch which was wrong.

3) For frequently accessed branches, the ability to access them in fewer cycles thereby improving the throughput of the branch prediction logic which in turn improves the overall throughput of the given microprocessor pipeline 300 of ~~Figure~~ FIG. 3, with the decoding [[e]] stage [[,]] 310, cycle cache address calculation stage [[,]] 320, cache access stage [[,]] 330, register access stage [[,]] 340, execute and branch resolution stage [[,]] 350, and register writeback stage [[,]] 360.

[0035] As illustrated generally in ~~Figure~~ FIG. 1, a Branch Target Buffer (BTB) layout of a BTB table 100 is an array relating to the branch prediction logic within a microprocessor. The BTB table 100 is responsible for the target prediction of a predicted taken branch. Given a 64 bit machine, per example, [[a]] the multi-associative BTB table 100 with associativity classes in columns A 120, B 121, C 122, and D 123 is fed some search address 110 of stated bits for a given range x:y where x>0, x<y, and y<63. This address is used to index a given row of the BTB table 100. For the given row, data is read out to corresponding data output lines 150, 151, 152, 153 from each of the corresponding associativity classes in columns A 120, B 121, C 122, and D 123 of the BTB table 100. What is contained within each entry such as entry 130, (~~Figure~~ in FIG. 1 [(),)] is shown in more detail, as an entry 200 in (~~Figure~~ FIG. 2 [[)]] of the BTB table 100, where there are three main pieces of data comprising: a branch tag 210, a target address 220, and a valid bit 230. The branch tag 210 is separated into two pieces, a high order range and a low order range. The high order range is used to compare with the high order search address bits. When there is a match, then the given entry has the potential of being a predicted branch. Besides the entry being valid, the low order range of the branch tag, y+1:63 must occur at or after the search address, as the goal is to find a predicted branch at or after the search address that is sent to the BTB table. Upon finding a match for a given entry, it is possible that there are multiple matches in each of the associativity [[set]] classes. Once again, the low order address bits are used to determine which associativity [[set]] class in columns 120, 121, 122, 123 in (~~Figure~~ FIG. 1 [[)]] comes first, while being at or after the initial search address 110. FIG. 3 illustrates one example of a microprocessor pipeline 300. When a match is found, the information is passed onto the main pipeline of the microprocessor where it will compare the predicted branch address to the address of the instruction(s) that are in the decoding [[e]] stage 310 of the microprocessor pipeline of FIG. 3. When a match between the two addresses is acquired, then the instruction decoding stage 310 is a predicted [taken] taken branch. In parallel upon the branch being found by the BTB table 100, a fetch request is placed in progress for the target address. Ideally the fetch request will have returned from the instruction cache by the time the branch has

decoded. Given the stated scenario, the target will be able to decode the cycle cache address stage 320 upon which the target of the branch is being computed; as the target has already been acquired via the cache. Had the branch not been predicted by the BTB table, a fetch request for the target would not have been able to have been made to the cache until the branch had decoded and the target was computed by the cycle cache address stage 320. Hence the prediction of the branch and the target has removed latency from the pipeline of the microprocessor.

[0036] When a branch is not predicted, a surprise branch 710, (FIG. 7) may be encountered, and it is to be written into the BTB table 100 in [[, ()] FIG. 1[())], and the BTB table 610 [[()] in FIG. 6[())] such that it can be predicted in respect to the next occurrence, upon branch resolution stage 350 of that branch at the execute time frame that the target of the branch and the direction of the branch resolution are known. It has been standard to use the known information at this time frame, per example, and write the branch into the BTB table 610. A branch can be a surprise branch 710 for one of two reasons: it was not in the queue, or it was in the queue but it was not found in time. In the latter ~~later~~ case, the branch should not ideally be added into the BTB table 610 again, as doing so would most likely replace some other good entry which is different from the duplicate entry that is to be written in to the BTB table 610. Through the usage timeline for write queue access and BTB writing timeline 500 in FIG. 5 of a BTB recent entry queue 400 in FIG. 4, or a BTB recent entry queue 620 in FIG. 6, whenever a new data in entry 140 in FIG. 1 is to be written during the send date to queue interval 510 into the BTB table 610, it is first compared with first and second compare equal comparators 430, 431, entry in the recent entry queue decision block 720 in FIG. 7 to the branch tag entries 420, 421 within the recent entry queue. If it matches as indicated by an output from OR gate 440 in FIG. 4, during the check for duplicate entry interval 520 in FIG. 5, one of the entries in the recent entry queue 620 then the data in entry 140, or data in entry 410 in FIG. 4 is blocked by block write 450 in FIG. 4 from being written into the BTB table 610 as it already exists somewhere within the BTB table 610. If the entry is not located within the recent entry

queue registers 420, 421, 620 then the entry is written during the write BTB write queue interval 530 into both the BTB table 610 and into the recent entry queue 620. The recent entry queue 620 works in a first data in 410, registers 420--first out registers 421 (FIFO) queue structure. ~~Such that~~ When a new entry is placed into the queue, the oldest entry in the queue is moved out to make room for the newest entry. Should an entry in the BTB table 610 be required to be invalidated for any reason, the recent entry queue registers 420, and registers 421 must be checked to determine if the entry is also contained within it. If the entry is in the recent entry queue registers 420, registers 421 and the entry is being invalidated in the BTB table 610, then the entry must also be invalidated in the recent entry queue registers 420, registers 421.

[0037] FIG. 6 illustrates one example of a recent entry queue modifying the BTB table hit detect logic. In FIG. 6 block 600 includes a BTB table array 610, a recent entry queue 620 and a Hit Detect-Compare Logic 630 which provides a hit detect output on line 640. ~~Figure 6 (600)~~ FIG. 6 illustrates an example of a recent entry queue 620 modifying the status of the BTB ~~status~~ table array 610. Since the recent entry queue 620 is a substantially smaller subset of the BTB table array 610, the ability to search for branch/target pairs in the recent entry queue 620 is significantly faster than searching in the far larger BTB table array 610. In the cycle, when the BTB table 610 is being accessed for a given row based on the search index, the recent entry queue 620 can be compared to the hit detect compare logic criteria 630 in parallel. Hence, whenever a new search is started, during the cycle when a read is being performed from the BTB table array 610, the recent entry queue 620 is doing a compare with the Hit Detect-Compare Logic 630 on its contents during the same cycle. The ability to do a hit detect 640, or search match, a cycle earlier improves the latency factor of the branch prediction logic for tight looping branches where the same branch is accessed repetitively and the BTB table array 610 by itself is unable to keep up because of the initial time required to access the BTB table array 610. As stated above, the recent entry queue 620 maintains a depth up to the associativity of the BTB table array 610

whereby while the BTB is indexed, the recent entry queue positions are input to comparison logic compare equal comparators 430,431 in FIG. 4 and hit detect compare logic 630 in FIG. 6. The recent entry queue depth is searched in respect to a matching branch in parallel with searching of the BTB output, where the hit detect compare logic 630 supports the associativity of the BTB table array 610. In searching the BTB table array 610 for the next predicted branch, the search strategy uses a subset of the recent entry queue as a subset of the BTB, table array 610 and preferably fast indexes recently encountered branches.

[0038] FIG. 7 illustrates one example of a state machine descriptor of a recent entry queue delaying decoding. In FIG. 7 block 700 ~~(700)~~ illustrates an example of a state machine description of a recent entry queue delaying decoding. [[e.]] As shown, because the BTB table 610 is working asynchronously from the decoding stage 310 of instructions in the microprocessor pipeline 300 of FIG. 3, it is possible for the pipeline to decode, with the decoding stage 310, a branch which is predictable, but was not yet found by the BTB table 610. In such cases, the branch is deemed as a surprise branch 710 and upon operation of the branch stage resolution stage 350, the execution cycle, of the branch, it will once again be placed into the BTB table 610. In the cases where this missed branch is a loop branch it will continue [[to]] not to be predicted for after each occurrence of not finding it, the branch prediction logic will restart based on the surprise branch decision block 710. In the case that the branch is already in the BTB table 610 as detected by OR gate 440 by the recent entry queue 620, recent entry queue decision block 720, and the branch is [[a]] taken as a branch from the resolved In a decision block 730 where the branch repeatedly occurs with a negative offset, in branch point, decision block 740, the recent entry queue 620 can be used to detect this scenario and cause the decoding 310 of microprocessor pipeline 300 after the branch to be delayed by the delay decode step 760 until the first prediction via the BTB table 610 is made such that the predicted branch address can be compared against all future decoding [[es]] of the BTB table 610. By causing a delay in the delay decode step 760 in the decoding [[e]] of the pipeline 300, the next iteration of the branch will be predicted by the BTB table 610 in time. Given that a BTB table 610 can have higher latency on start-up compared to the latency thereof once it is running, the one time delay [[of]] by the decode in normal fashion step 750, or the delay decode block 760 of decoding [[e]] can be enough to allow the branch to be predicted for all future iterations of the current looping pattern.